Software Engineering

(7) Maintenance / Management

Sokendai / National Institute of Informatics
Fuyuki Ishikawa / 石川　冬樹
f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

TOD

- **<u>Maintenance</u>**
- Refactoring
- Project Management

Maintenance: Overview

- Maintenance（保守）

activities to maintain and manage the developed software after release

  - Different from hardware that has aging and is difficult to modify
  - Known to have a very large cost
    - 67% of the whole cost (1994) [Schach, The economic impact of software reuse on maintenance, 1994]
    - 5-year cost is double of the initial development fee (2020, Japan)

[日本情報システム・ユーザー協会, ソフトウェアメトリックス調査2020 システム開発・保守調査, 2020]

Maintenance: Classification

- Types of maintenance
  - Corrective（修正）: fix bugs
  - Adaptive （適応）: adapt to changes in operational environments
  - Perfective（改善）: improve manageability or reusability (specifically, refactoring activities)
  - Preventive （予防）: prepare for potential issues in future use

Maintenance: Challenges

- The most significant and difficult activities: tracing and understanding to identify necessary tasks
  - e.g., what goals each requirement contributes to
  - e.g., what requirements each function supports
  - e.g., why this architecture was obtained
  - e.g., what tests were conducted for each element
  - …

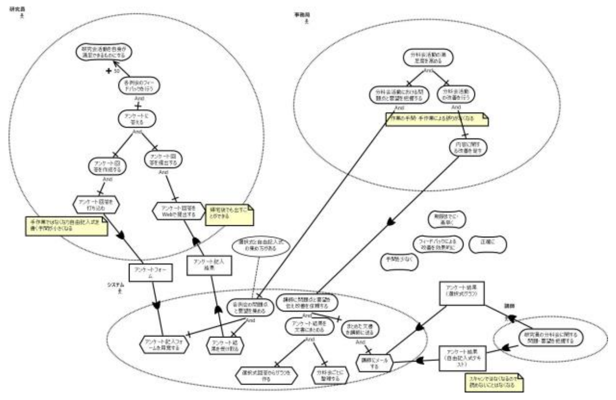
➡ Change Impact Analysis（変更影響分析）

Maintenance: Challenges

- Difficulties in reading code of other people or code I wrote a few months ago
- No assumption on availability of the same team for development and maintenance
  - Sometimes feasible if the customer wants a contract like "update every 3 months"
  - Otherwise, the development team members will be allocated to other new projects, probably not available when the maintenance is triggered

Change Impact Analysis: Forward Approach

- Approach (1): record and keep necessary information
  - Rigorous models in this class will be useful in maintenance
  - Traceability （追跡可能性） is the key based on dependency
    - e.g., goal models will allow to identify which super-/sub-goals are affected by change in a goal
  - Traceability Matrix: keep traceability between requirements and implementations/test cases



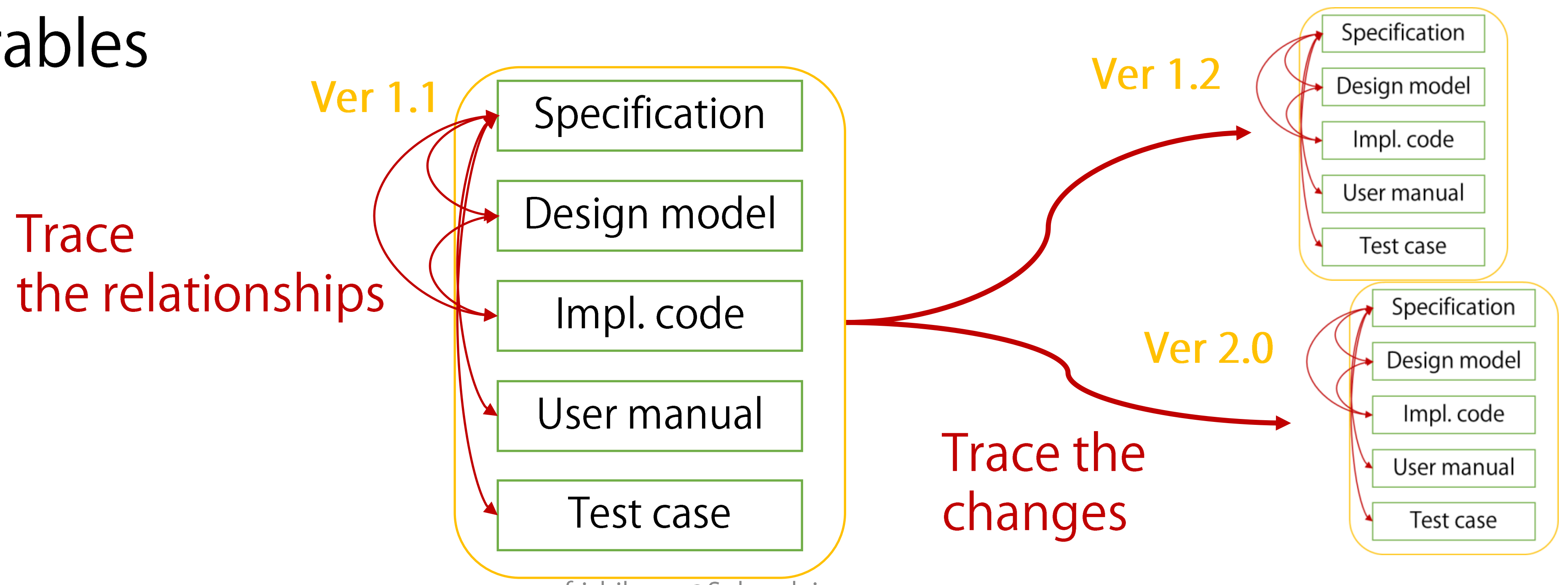
	REQ1.1	REQ1.2	REQ2.1
Component 01	X		
Component 02	X	X	
Component 03			X

Change Impact Analysis: Backward Approach

- Approach (2): Reverse Engineering
  - Extract information from the implementation code
    - When upper-level documents are missing or obsolete
    - e.g., generate class diagrams from code
    - e.g., extract control and data flow from code
    - e.g., search for elements in code or models
  - More advanced recently with techniques for natural language processing and data mining
    - e.g., predict links between words in natural-language documents and names in code

Relevant Topic: Configuration Management

- Configuration Management（構成管理）
  - Trace and manage changes in software systems
  - Wider than "version management" with focus on the whole consistency of elements including documents and other deliverables



Relevant Topic: Regression Testing (Revisited)

- 回帰テスト（Regression Testing）
  - Check whether something is worse than the previous version
  - Background: it is very typical that a fix for a certain function leads to failures in other parts
    (Japanese engineers often say 「デグレ」 for degradation)

Relevant Topic: Regression Testing (Revisited)

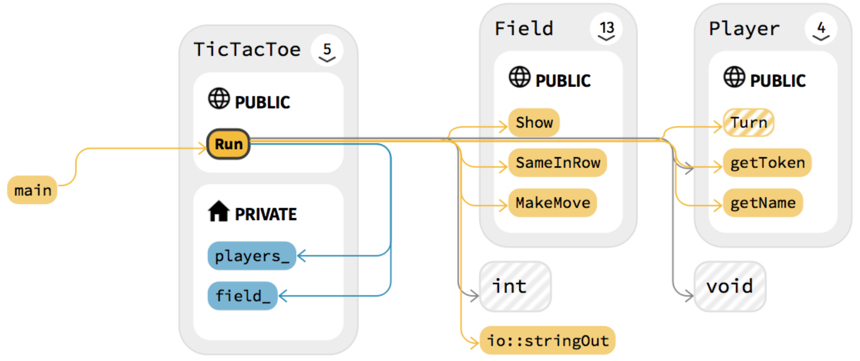
- Change impact analysis on tests
  1. No need to execute the test again as the test target is not affected by the changes
  2. Need to execute the test again as the test target may be affected by the changes to check if the same test passes
  3. Abandon the test case as it is no more a valid test after the change, e.g., the expected outcome changed

Typically, wrong judgement of #2 as #1 by hidden dependencies

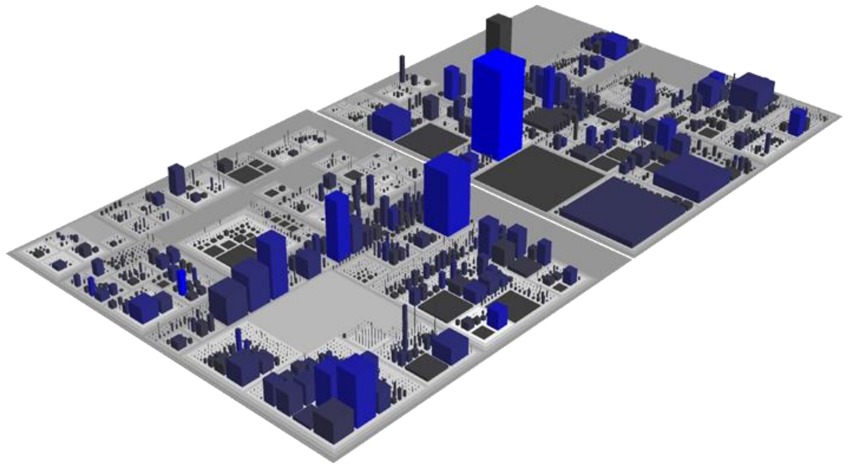
Related Topic: Visualization

- e.g., Sourcetrail
  - Visualization of method invocation and file inclusion

[https://www.sourcetrail.com/]



- e.g., CodeCity
  - Metrics over different modules are visualized as a city as well as their evolution

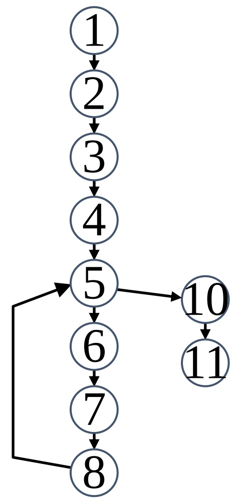


[https://wettel.github.io/codecity.html]

Relevant Topic: Dependency Analysis of Code

- Control Flow Graph
  - Control dependency: it depends on conditions in S0 whether S1 is executed or not
  - Data dependency: variable values used in S1 depend on definitions or assignments in S0

```
 1: int func(int data[], int n) {
 2: int sum = 0;
 3: int prod = 1;
 4: int i = 0;
 5: while (i < n) {
 6:   sum = sum + data[i];
 7:   prod = prod * data[i];
 8:   i = i + 1;
 9: }
10: print(sum);
11: print(prod);
```



e.g.,
line 6 has
- control dependency on line 5
- data dependency on line 2

Relevant Topic: Slicing

- Slicing

  Extract parts of the code relevant with the current target of analysis

  - i.e., parts that the target depends on directly or indirectly

Slicing for line 10

```
 1: int func(int data[], int n) {
 2: int sum = 0;
 3: int prod = 1;
 4: int i = 0;
 5: while (i < n) {
 6:   sum = sum + data[i];
 7:   prod = prod * data[i];
 8:   i = i + 1;
 9: }
10: print(sum);
11: print(prod);
```

```
 1: int func(int data[], int n) {
 2: int sum = 0;

 4: int i = 0;
 5: while (i < n) {
 6:   sum = sum + data[i];

 8:   i = i + 1;
 9: }
10: print(sum);
```

TOD

- Maintenance
- Refactoring
- Project Management

Refactoring: Definition

- Perfective/preventive maintenance does not make any change on the existing functions

➡ Refactoring （リファクタリング）：
make changes in architecture or implementation methods without changing the functions

  - Sometime called reengineering or rejuvenation

Refactoring: Significance

- Difficult to predict future changes
  - Architecture patterns or design patterns aim at supporting specific types of changes or reuses
- Difficult to have “clean code that works”
  - “Clean code” in your mind often does not work
  - You may make it work, then you have “dirty code”
  - Some methods rather recommend this way of “first dirty working code, then refactoring” for complex functions (e.g., test-driven development, to be discussed next week)

Refactoring: Triggers

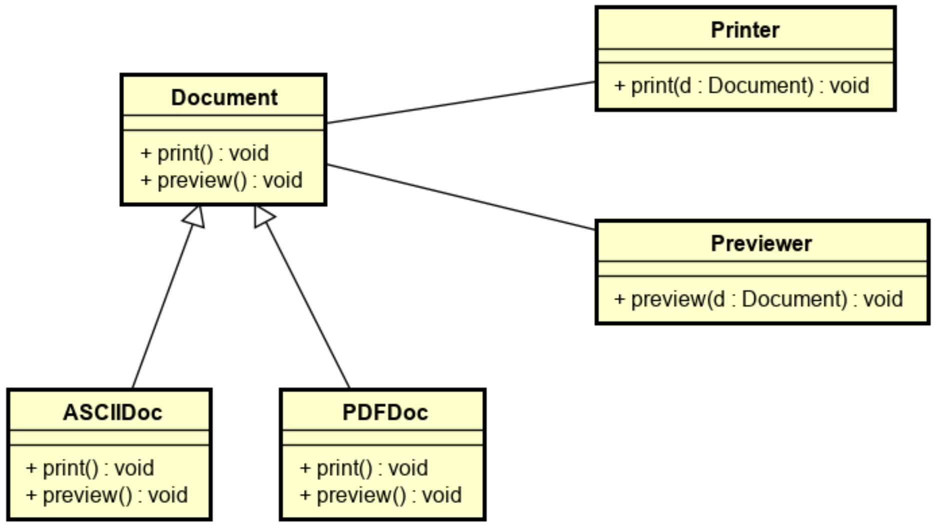
- Bad Smells: trends that imply potential issues
  - Duplicated Code: similar code in multiple places
  - ➡ Extraction of methods or classes
  - Large Class: many responsibilities in one class
  - ➡ Extraction of classes/subclasses or replacement of multiple data values by an object
  - Feature Envy: a method interacting more with other classes
  - ➡ Move of fields or methods

Refactoring: Practice

- Realize as a combination of small operations that does not break the current code
- Now IDE (Integrated Development Environment) has strong supports for refactoring
  - Renaming of class, field, or method
  - Extracting a specified part of code as a new method
  - Moving fields or methods to the super-class
  - Moving static fields or methods to another class
  - …

Refactoring: Example

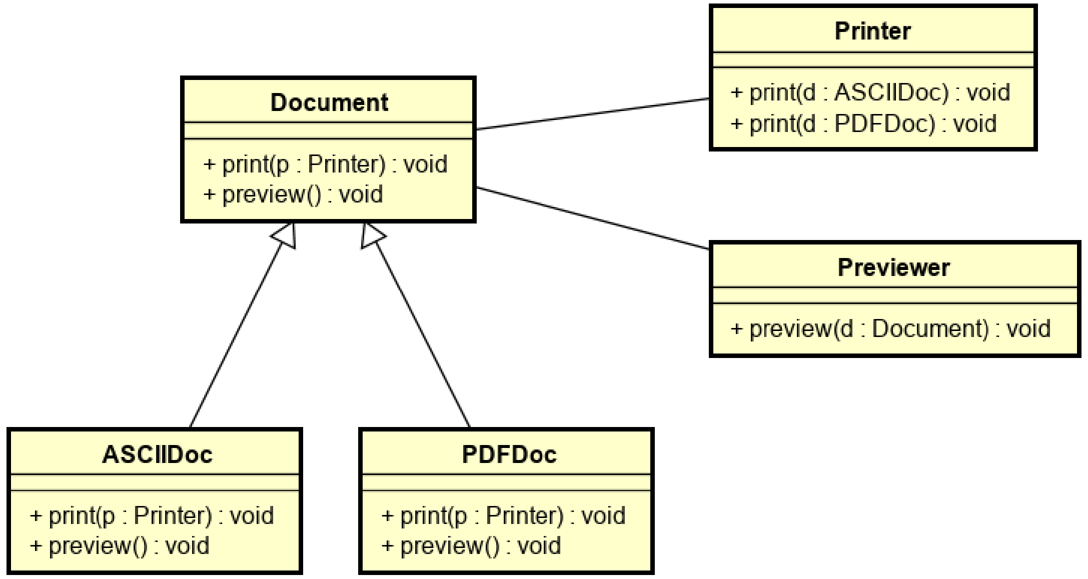
- Initial design
  - The document classes (XXDoc) own the responsibilities for print/view
    - The Printer/Previewer classes only call the methods provided by XXDoc
  - This is good when we have more XXDoc but only with print/view
- Change
  - We don't have more XXDoc but we expect more ways of processing (convert/scan/etc.)



Refactoring: Example

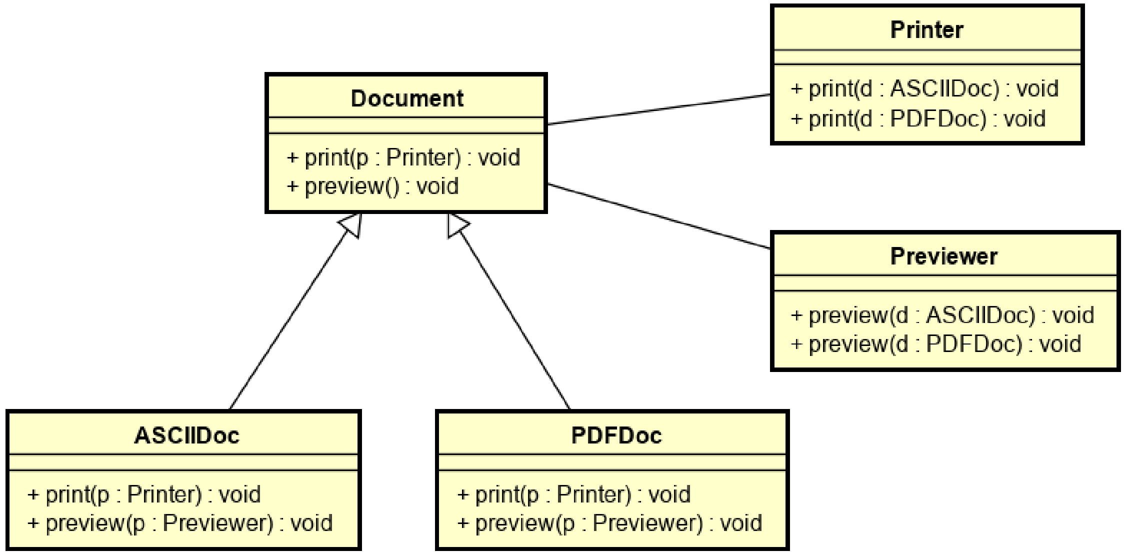
- First, move the print behavior from the XXDoc classes to the Printer class

➡ Now, the print methods in XXDoc just call the Printer object


Document
+ print(p : Printer) : void
+ preview() : void
Printer
+ print(d : ASCIIDoc) : void
+ print(d : PDFDoc) : void
Previewer
+ preview(d : Document) : void
ASCIIDoc
+ print(p : Printer) : void
+ preview() : void
PDFDoc
+ print(p : Printer) : void
+ preview() : void

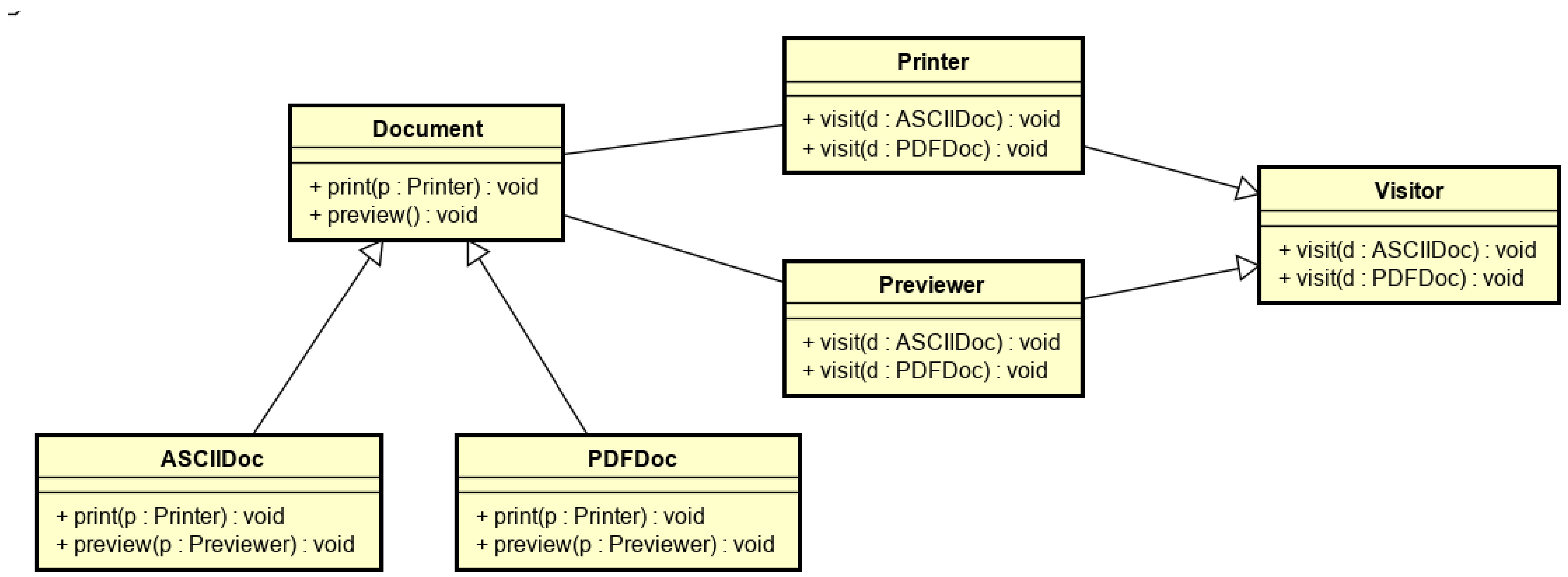
Refactoring: Example

- Do the same for the preview function


Document
+ print(p : Printer) : void
+ preview() : void
Printer
+ print(d : ASCIIDoc) : void
+ print(d : PDFDoc) : void
Previewer
+ preview(d : ASCIIDoc) : void
+ preview(d : PDFDoc) : void
ASCIIDoc
+ print(p : Printer) : void
+ preview(p : Previewer) : void
PDFDoc
+ print(p : Printer) : void
+ preview(p : Previewer) : void

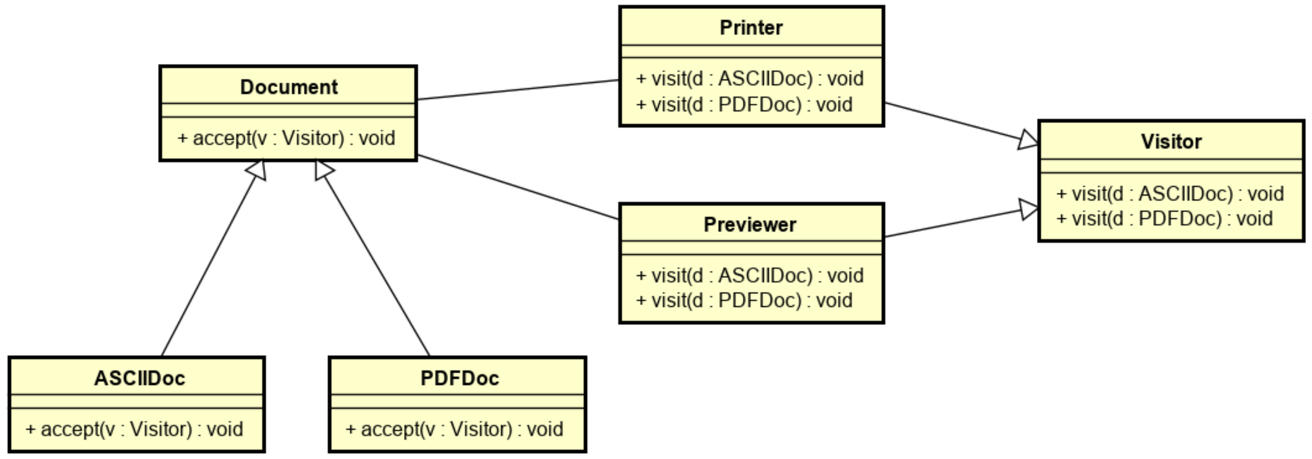
Refactoring: Example

- Unify the Printer/Previewer interfaces


Document
+ print(p : Printer) : void
+ preview() : void
Printer
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
Previewer
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
Visitor
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
ASCIIDoc
+ print(p : Printer) : void
+ preview(p : Previewer) : void
PDFDoc
+ print(p : Printer) : void
+ preview(p : Previewer) : void

Refactoring: Example

- Last: update the XXDoc classes to use the unified interface


Document
+ accept(v : Visitor) : void
Printer
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
Visitor
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
Previewer
+ visit(d : ASCIIDoc) : void
+ visit(d : PDFDoc) : void
ASCIIDoc
+ accept(v : Visitor) : void
PDFDoc
+ accept(v : Visitor) : void

Note: Visitor Pattern

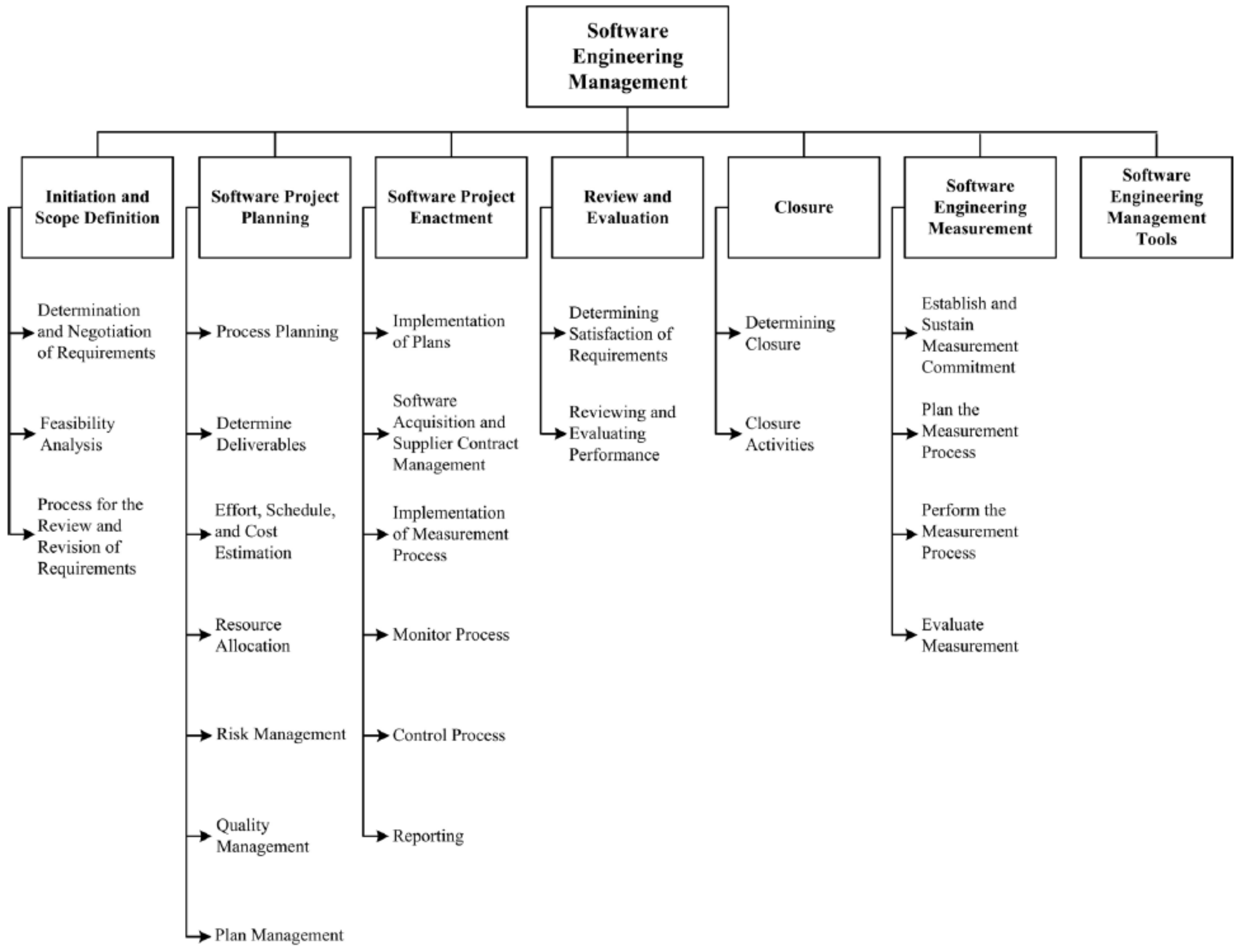
- The resulting design is the Visitor pattern in GoF
  - Data class implements a procedure to traverse the data structure
  - Concrete processing functions are implemented as visitors and passed to the data class
- e.g., for a program code
  - We prepare a component to traverse all of the nodes of the abstract syntax tree
  - Then, we can pass any processing function to it, e.g., counting elements, checking consistency, etc.

TOD

- Maintenance
- Refactoring
- <u>Project Management</u>

Management

- Body of knowledge from SWEBOK 3.0


Software Engineering Management
Initiation and Scope Definition
Determination and Negotiation of Requirements
Feasibility Analysis
Process for the Review and Revision of Requirements
Software Project Planning
Process Planning
Determine Deliverables
Effort, Schedule, and Cost Estimation
Resource Allocation
Risk Management
Quality Management
Plan Management
Software Project Enactment
Implementation of Plans
Software Acquisition and Supplier Contract Management
Implementation of Measurement Process
Monitor Process
Control Process
Reporting
Review and Evaluation
Determining Satisfaction of Requirements
Reviewing and Evaluating Performance
Closure
Determining Closure
Closure Activities
Software Engineering Measurement
Establish and Sustain Measurement Commitment
Plan the Measurement Process
Perform the Measurement Process
Evaluate Measurement
Software Engineering Management Tools

Example of Management Method: Cost Estimation

- Functional Size Measurement（機能規模測定）
  - Methods such as COCOMO, COSMIC, NESMA
- Common concept of “function point”:
  we derive points or scores for each function by counting inputs and outputs and evaluating the complexity
  - e.g., IN 24 × Complexity 4 ＋ OUT 14 × Complexity 5 ＋ ⋯ ＝ 320
  - May consider factors of complexity such as “high performance required” or “inputs made in multiple views”
  - May consider factors such as experience levels of engineers

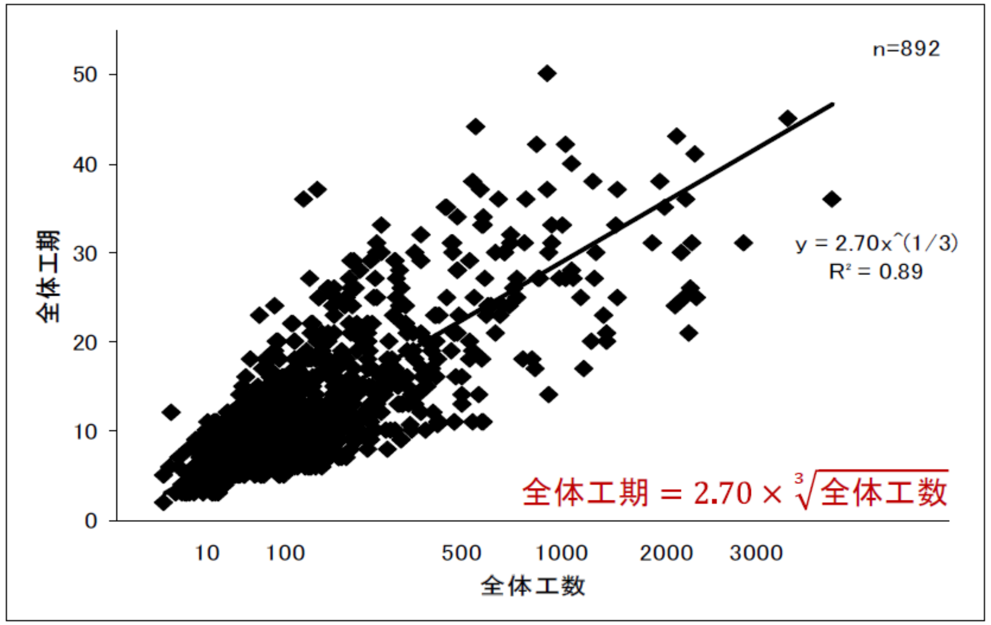
Example of Management Method: Software Equation

- $E = L^3/P^3t^4$
  - Necessary effort (person-year) E is
  - Proportional to the cube of the code size L
  - Inversely proportional to the biquadratic of the time t (year)
  - Inversely proportional to the cube of productivity P
- Example: $L = 33{,}000, P = 12{,}000$
  - $t = 1.75$ (year) → $E = 3.8$ (man-year)
  - $t = 1.3$ (year) → $E = 7.2$ (man-year)
- There is a lower-bound of the time

Double of persons to decrease the period by about half a year

Example of Management Method: Simple Practice

- One example of guideline/statistics in Japan
  - Num of views → effort estimation
    - $E = 1.91 \times \left(\#views + \frac{2}{3}\#dataschemes\right)$
  - Then, effort → time estimation
    - $T = 2.7\sqrt[3]{E}$
  - Simple, even done during meetings



Cited from [日本情報システム・ユーザー協会, ソフトウェアメトリックス調査2020 システム開発・保守調査, 2020]

Metrics

- Measurement and monitoring of metrics （メトリクス）を is essential for management
    - Size of development
    - Progress and effectiveness of tests
    - Design quality
    - Source code quality
    - …

Example of Metrics (1)

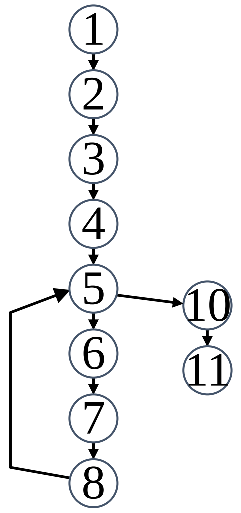
- Size of development
  - LOC (Line of Code) / KLOC: understand the size of code after development by counting the lines with standard formatting
  - Function Point: estimate the size of development beforehand
- Progress and effectiveness of tests
  - Test Case Density and Defect Density: count test cases or detected defects per 1 KLOC/1 FP to evaluate test adequacy and product quality

Example of Metrics (2)

- Design quality
  - [Card, 1990]
    - Fan-out $f_{out}(i)$: num of modules directly invoked by the module i
    - Structural complexity $S(i) = f_{out}^2(i)$
    - Data complexity $D(i) = v(i)/f_{out}(i) + 1$
      where $v(i)$ is the num of input/output variables
    - System complexity $C(i) = S(i) + D(i)$
  - Cohesion and coupling
    - e.g., num of references to elements in other classes
    - e.g., num of methods in the same class that use a certain field

Example of Metrics (3)

- Source code quality
  - Cyclomatic complexity（サイクロマティック複雑度）：
    measuring the complexity by counting independent paths
    “#edges - #nodes + 2“ in the control graph
    - Empirically, <=10 is desirable and
      >=30 leads to many defects



10 – 11 + 2 = 1

Summary

- Maintenance
  - Very significant but difficult activities
  - Practical solutions investigated by combining prior documentation and by posterior analysis and mining
- Management
  - Deals with cost and process aspects based on statistics and data